Programmation fonctionnelle avec OCaml

1ère séance, 19 février 2015

- 6 séances de 1h30 de cours et 3h de TP
- 3 projets avec soutenance
- D'autres transparents sont disponibles avec vidéo (intranet)

Samuel Hornus : samuel.hornus@inria.fr

http ://www.loria.fr/~shornus/ocaml/

La boucle interactive

On lance l'interpréteur dans un terminal :
$ ocaml ($ rlwrap ocaml)

1. On écrit une **expression** :
   ```
   # expression ;;
   ```
2. L'interpréteur **évalue** l'expression et **imprime** sa valeur :
   ```
   réponse
   ```

La boucle interactive

On lance l'interpréteur dans un terminal :

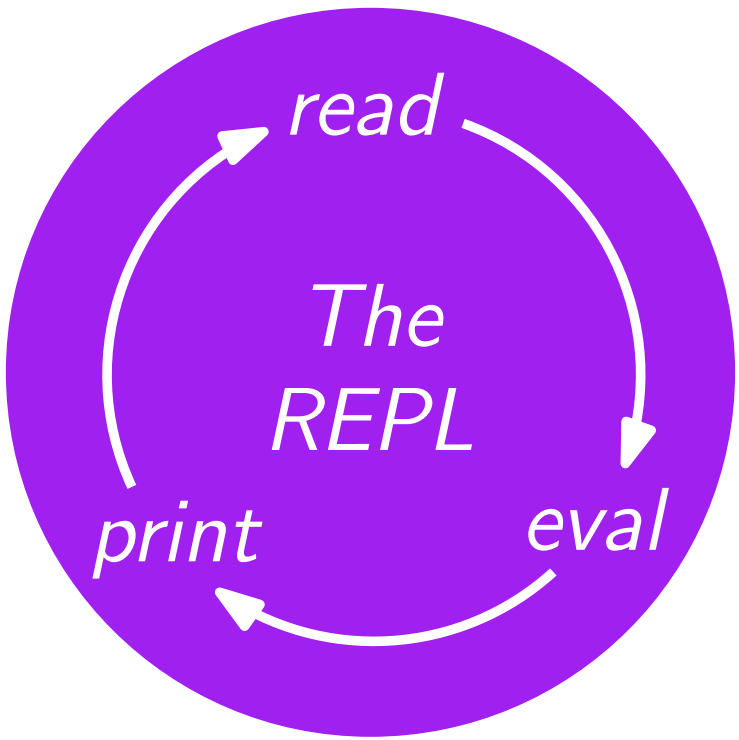
$ ocaml ($ rlwrap ocaml)

1. On écrit une **expression** :

   `# expression ;;`

2. L'interpréteur **évalue** l'expression et **imprime** sa valeur :

   réponse



Pour terminer la session :

`# #quit;;` ou `^D`

Quelques expressions simples

```
# 23;;

# 'f';;

# -24;;

# 10.45;;

# (87, 'R');;

# "fa sol la";;
```

Quelques expressions simples

```
# 23;;
- : int = 23
# 'f';;
- : char = 'f'
# -24;;
- : int = -24
# 10.45;;
- : float = 10.45
# (87, 'R');;
- : int * char = (87, 'R')
# "fa sol la";;
- : string = "fa sol la"
```

Chaque expression possède

- un **type**
- une **valeur**

Quelques expressions un peu moins simples

```
# 23+10;;
- : int = 33
# ['f'; 'a'; '4'];;
- : char list = ['f'; 'a'; '4']
# 24 / 13;;
- : int = 1
# 10.45 *. 10.0;;
- : float = 104.5
# cos;;
- : float -> float = <fun>
# cos 3.14159;;
- : float = -0.99999999996479261
# String.length "fa sol la";;
- : int = 9
```

Quelques expressions un peu moins simples

```
# 23+10;;
- : int = 33
# ['f'; 'a'; '4'];;
- : char list = ['f'; 'a'; '4']
# 24 / 13;;
- : int = 1
# 10.45 *. 10.0;;
- : float = 104.5
# cos;;
- : float -> float = <fun>
# cos 3.14159;;
- : float = -0.99999999996479261
# String.length "fa sol la";;
- : int = 9
```

Une **liste** de caractères

Quelques expressions un peu moins simples

```
# 23+10;;
- : int = 33
# ['f'; 'a'; '4'];;
- : char list = ['f'; 'a'; '4']
# 24 / 13;;
- : int = 1
# 10.45 *. 10.0;;
- : float = 104.5
# cos;;
- : float -> float = <fun>
# cos 3.14159;;
- : float = -0.99999999996479261
# String.length "fa sol la";;
- : int = 9
```

Une **liste** de caractères

Observer le nom de l'opérateur : *.

Quelques expressions un peu moins simples

```
# 23+10;;
- : int = 33
# ['f'; 'a'; '4'];;
- : char list = ['f'; 'a'; '4']
# 24 / 13;;
- : int = 1
# 10.45 *. 10.0;;
- : float = 104.5
# cos;;
- : float -> float = <fun>
# cos 3.14159;;
- : float = -0.99999999996479261
# String.length "fa sol la";;
- : int = 9
```

Une **liste** de caractères

Observer le nom de l'opérateur : *.

Application d'une fonction à un argument.

Liaison d'une valeur à un nom

```
# let a = 42;;
val a : int = 42

# let b = a - 20;;
val b : int = 22



# b+a;;
- : int = 64
```

Liaison d'une valeur à un nom

```
# let a = 42;;
val a : int = 42

# let b = a - 20;;
val b : int = 22
```

Syntaxe :

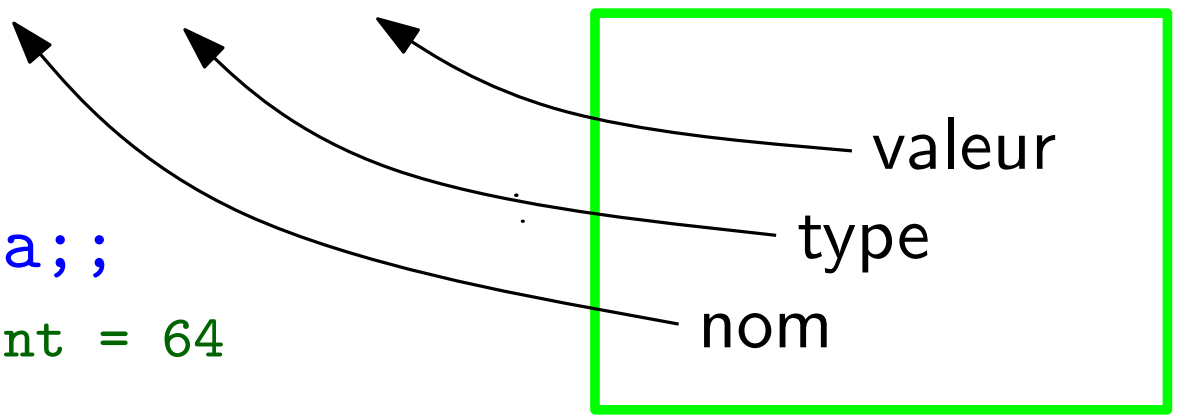
`let` *nom* `=` *expression* `;;`

liaison

```
# b+a;;
- : int = 64
```

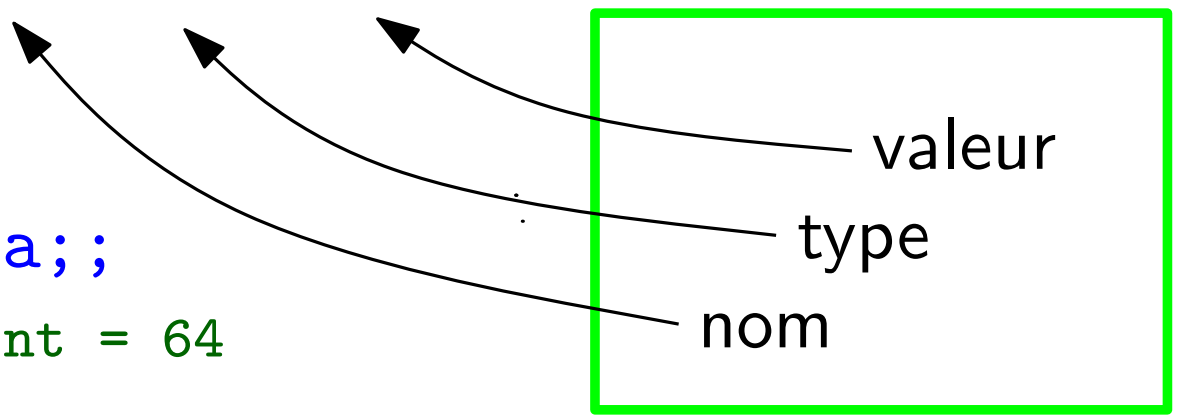
Liaison d'une valeur à un nom

```
# let a = 42;;
val a : int = 42

# let b = a - 20;;
val b : int = 22

# b+a;;
- : int = 64
```



Liaison d'une valeur à un nom

```
# let a = 42;;
val a : int = 42

# let b = a - 20;;
val b : int = 22

# b+a;;
- : int = 64
```



- La liaison d'un nom à une valeur est enregistrée dans un **environnement**.
- Plusieurs environnements existent souvent en même temps.

Liaison d'une valeur à un nom (suite)

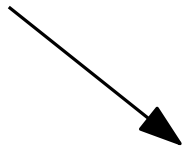
```
# let b = false;;
val b : bool = false

# c;;
Error: Unbound value c

# let l = [67;-4;100];;
val l : int list = [67; -4; 100]

# let first = List.hd l;;
val first : int = 67
```

L'expression conditionnelle

`if` *condition* `then` *expression1* `else` *expression2*

L'expression conditionnelle

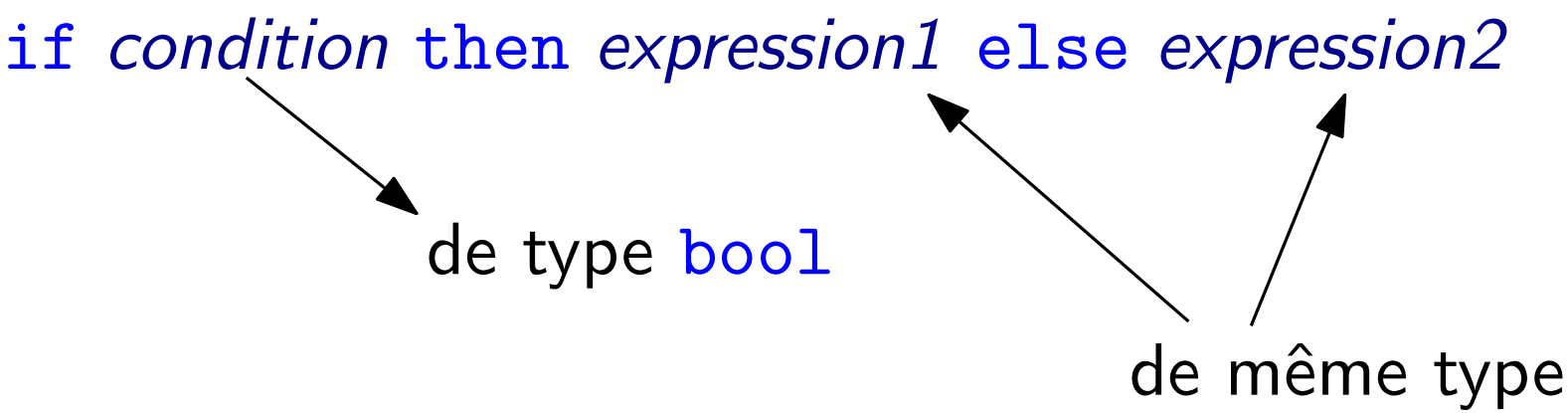
`if` *condition* `then` *expression1* `else` *expression2*



de type `bool`

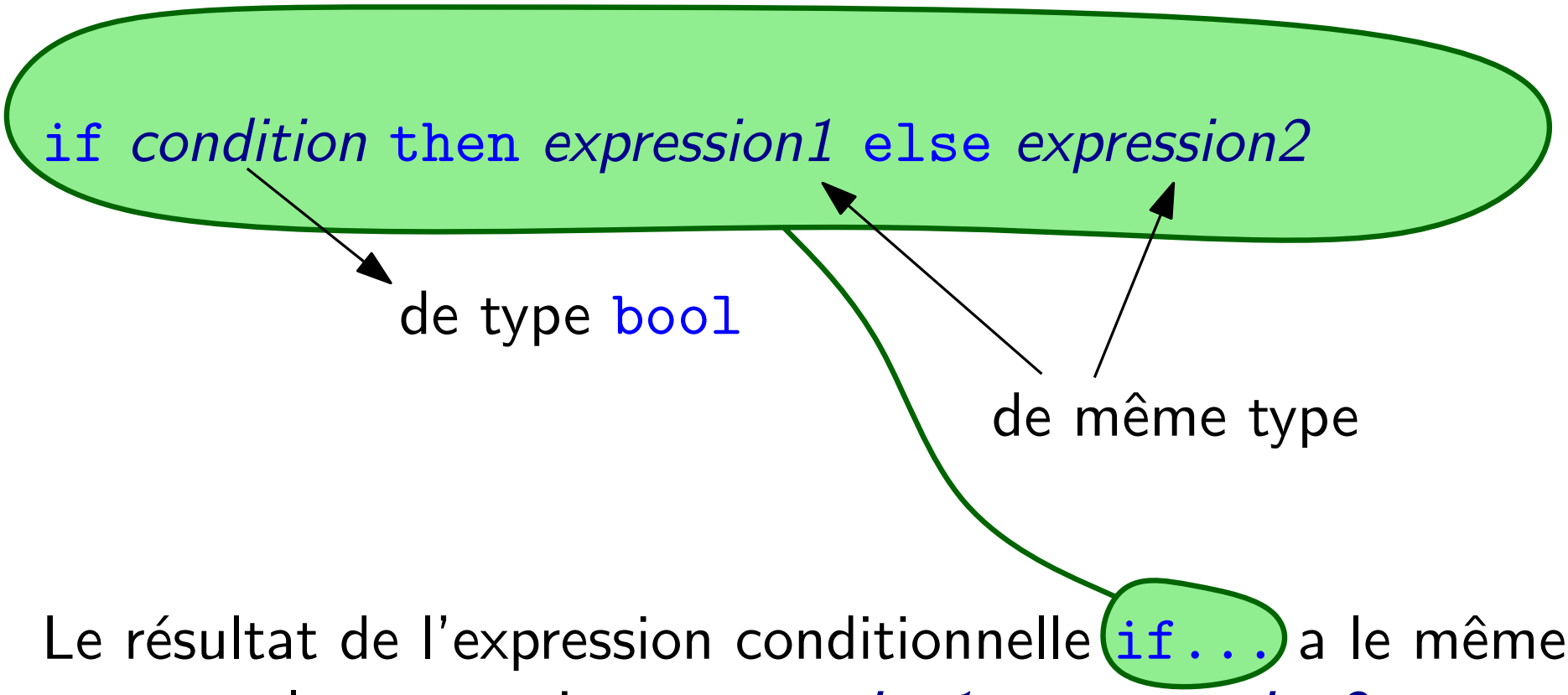
Opérateurs booléens : `not`, `||`, `&&`

L'expression conditionnelle



Opérateurs booléens : `not`, `||`, `&&`

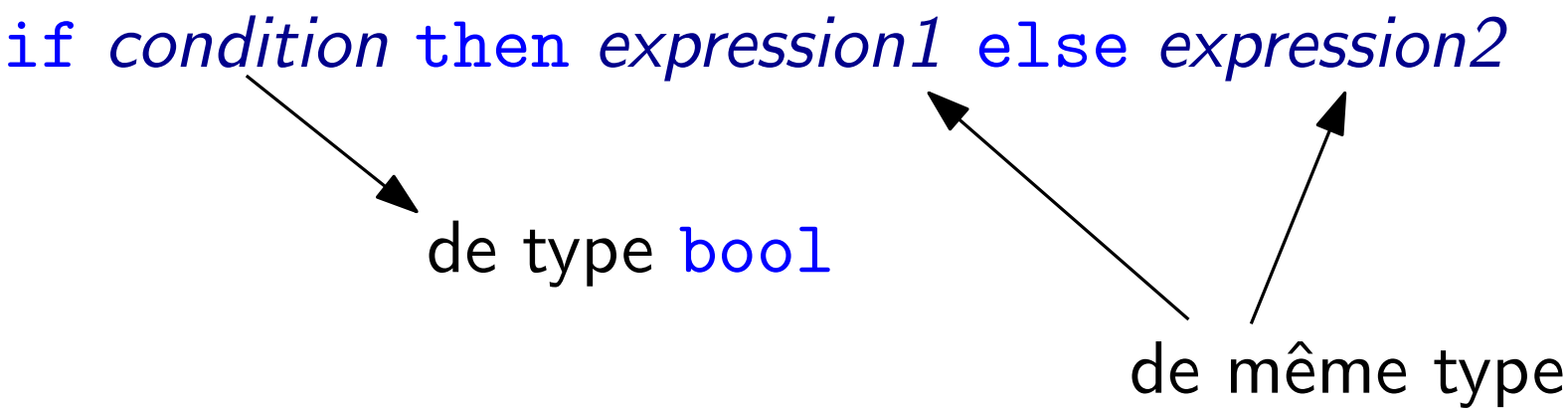
L'expression conditionnelle



Le résultat de l'expression conditionnelle `if...` a le même type que les expressions *expression1* et *expression2*

Opérateurs booléens : `not`, `||`, `&&`

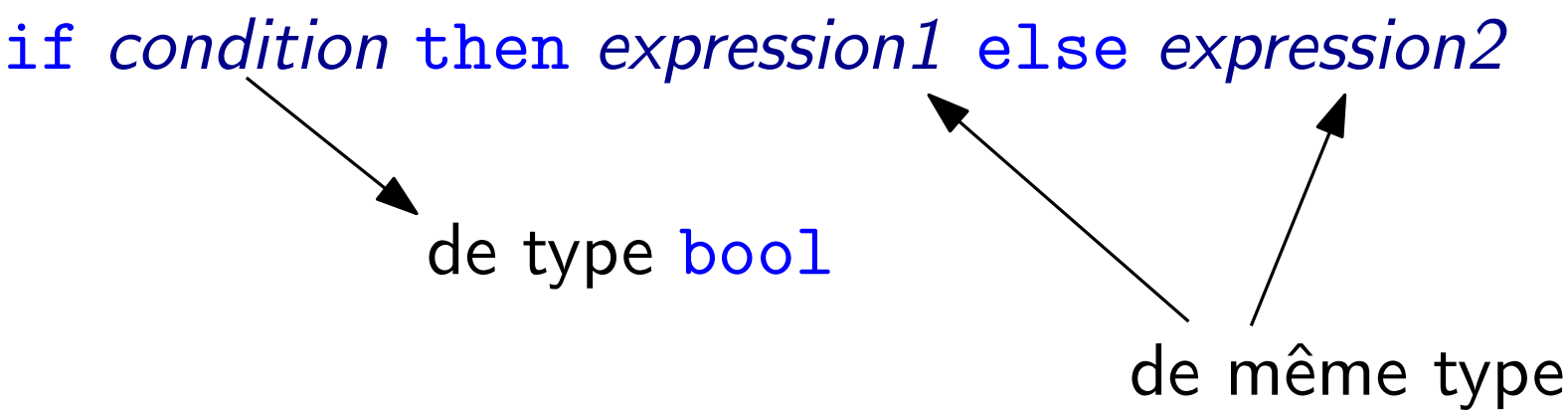
L'expression conditionnelle




```
# if 22 >= 21 then "gagné !"
                  else "perdu !";;
- : string = "gagné !"
```

Opérateurs booléens : `not`, `||`, `&&`

L'expression conditionnelle




```
# if 0 < 1 then 42 else 42.0;;
Error: This expression has type float but an expression was
expected of type int
```

Opérateurs booléens : `not`, `||`, `&&`

Filtrage par motifs – *pattern matching*

On doit souvent « calculer » une valeur en fonction d'une autre :

```
# let x = if c = 0 then "zéro" else
          if c = 1 then "un" else
          if c = 2 then "deux" else
          ...
          if c = 7 then "sept" else
          if c = 8 then "huit" else
          if c = 9 then "neuf" else
          failwith "Problème !";;
val x : string = ...
```

Filtrage par motifs – *pattern matching*

Une écriture plus concise et plus puissante :

```
# let x = match c with
          | 0 -> "zéro"
          | 1 -> "un"
          | 2 -> "deux"
          ...
          | 7 -> "sept"
          | 8 -> "huit"
          | 9 -> "neuf"
          | _ -> failwith "Problème !";;
val x : string = ...
```

Syntaxe du filtrage par motifs

```
match expr with
        | motif_1 -> expression_1
        | motif_2 -> expression_2
        | motif_3 -> expression_3
        | ...
```

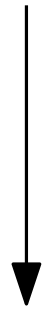
← la *valeur* à filtrer

le filtrage : une suite de filtres

Syntaxe du filtrage par motifs

```
match expr with
        | motif_1 -> expression_1
        | motif_2 -> expression_2
        | motif_3 -> expression_3
        | ...
```

← la *valeur* à filtrer



le filtrage : une suite de filtres

Un motif est la description d'un ensemble de valeur(s).
On peut le voir comme un *moule* dans lequel un valeur *rentre* ou *ne rentre pas*

Syntaxe du filtrage par motifs

```
match expr with ← la valeur à filtrer
        | motif_1 -> expression_1
        | motif_2 -> expression_2
        | motif_3 -> expression_3
        | ...
```

le filtrage : une suite de filtres

Un motif est la description d'un ensemble de valeur(s).
On peut le voir comme un *moule* dans lequel un valeur *rentre* ou *ne rentre pas*

Si la valeur de `expr` correspond au `motif_i` alors le filtrage `match..` prend pour valeur celle de l'`expression_i`

Syntaxe du filtrage par motifs

```
match expr with ← la valeur à filtrer
        | motif_1 -> expression_1
        | motif_2 -> expression_2
        | motif_3 -> expression_3
        | ...
```

le filtrage : une suite de filtres

Un motif est la description d'un ensemble de valeur(s).
On peut le voir comme un *moule* dans lequel un valeur *rentre* ou *ne rentre pas*

Si la valeur de `expr` correspond au `motif_i` alors le filtrage `match..` prend pour valeur celle de l'`expression_i`

Intuitivement, le filtrage parcours les motifs de haut en bas et s'arrête au premier motif qui correspond à la valeur filtrée.

Syntaxe du filtrage par motifs

Exemple :

```
# let aire p = match p with
    | ("carré", c) -> c *. c
    | ("disque", r) -> 3.1419 *. r *. r
    | ("équi", r) -> (sqrt 3.0) /. 2.0 *. r *. r
    | (s, _) -> failwith ("Qu'est-ce qu'un "^s^" ?");;
val aire : string * float -> float = <fun>
```

Syntaxe du filtrage par motifs

Exemple :

```
# let aire p = match p with
    | ("carré", c) -> c *. c
    | ("disque", r) -> 3.1419 *. r *. r
    | ("équi", r) -> (sqrt 3.0) /. 2.0 *. r *. r
    | (s, _) -> failwith ("Qu'est-ce qu'un "^s^" ?");;
val aire : string * float -> float = <fun>

 # aire ("équi", 2.0);;
 - : float = 3.46410161513775439
 # aire ("carré", 2.0);;
 - : float = 4.
 # aire ("rond", 1.0);;
 Exception: Failure "Qu'est-ce qu'un rond ?".
```

Fonctions

Syntaxe : `function` *paramètre* `->` *expression*

```
# function x -> x+1;;
- : int -> int = <fun>
```

Application de la fonction à l'argument 27 :

```
# (function x -> x+1) 27;;
- : int = 28
```

Nommer la fonction dans l'environnement courant :

```
# let maFonc = function x -> x+1;;
val maFonc : int -> int = <fun>
# maFonc 31;;
- : int = 32
```

Fonctions

Fonction à 2 paramètres ?

```
# let ajoute = function x -> function y -> x +. y;;
val ajoute : float -> float -> float = <fun>
```

Application de la fonction :

```
# ajoute 6.6 3.4;;
- : float = 10.
```

Une syntaxe plus courte :

```
# let ajoute = fun x y -> x +. y;;
val ajoute : float -> float -> float = <fun>
```

Fonctions

Encore plus court :

```
# let ajoute x y = x +. y;;
val ajoute : float -> float -> float = <fun>
```

Idem pour plus de paramètres :

```
# let calcul a b c d = a*a+b - ( c/d );;
val calcul : int -> int -> int -> int -> int = <fun>
```

Fonctions

Encore plus court :

```
# let ajoute x y = x +. y;;
val ajoute : float -> float -> float = <fun>
```

Idem pour plus de paramètres :

```
# let calcul a b c d = a*a+b - ( c/d );;
val calcul : int -> int -> int -> int -> int = <fun>
```

À retenir :

```
function x -> function y -> exp
```

est identique à

```
fun x y -> exp
```

```
let maFonction = fun x y -> exp
```

est identique à

```
let maFonction x y = exp
```

Exemple de fonction récursive : la factorielle

Rappel : $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1.$

Exemple de fonction récursive : la factorielle

Rappel : $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1.$

```
# let facto n = if n <= 1 then 1
                        else n * facto (n-1);;
Error: Unbound value facto
```

Exemple de fonction récursive : la factorielle

Rappel : $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1.$

```
# let facto n = if n <= 1 then 1
                        else n * facto (n-1);;
Error: Unbound value facto

# let rec facto n = if n = 1 then 1
                        else n * facto (n-1);;
val facto : int -> int = <fun>
```

Exemple de fonction récursive : la factorielle

Rappel : $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1.$

```
# let facto n = if n <= 1 then 1
                        else n * facto (n-1);;
Error: Unbound value facto

# let rec facto n = if n = 1 then 1
                        else n * facto (n-1);;
val facto : int -> int = <fun>

# facto 12;;
- : int = 479001600

# List.map facto [0;1;2;3;4;5;6;7;8;9];;
- : int list = [1; 1; 2; 6; 24; 120; 720; 5040; 40320; 362880]
```

Exemple de fonction récursive : la factorielle

Rappel : $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1.$

```
# let facto n = if n <= 1 then 1
                        else n * facto (n-1);;
Error: Unbound value facto

# let rec facto n = if n = 1 then 1
                        else n * facto (n-1);;
val facto : int -> int = <fun>

# facto 12;;
- : int = 479001600

# List.map facto [0;1;2;3;4;5;6;7;8;9];;
- : int list = [1; 1; 2; 6; 24; 120; 720; 5040; 40320; 362880]
```

Une fonction est une valeur, comme un entier ou une liste.

```
# let ajouteur x = (function s -> s + x);;
val ajouteur : int -> int -> int = <fun>

# let t = ajouteur 3;;
val t : int -> int = <fun>

# t 10;;
- : int = 13

# let compose f g = (function x -> f (g x));;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

# let mystere = compose t (ajouteur 2);;
val mystere : int -> int = <fun>
```

Une fonction est une valeur, comme un entier ou une liste.

```
# let ajouteur x = (function s -> s + x);;
val ajouteur : int -> int -> int = <fun>

# let t = ajouteur 3;;
val t : int -> int = <fun>

# t 10;;
- : int = 13

# let compose f g = (function x -> f (g x));;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

# let mystere = compose t (ajouteur 2);;
val mystere : int -> int = <fun>

# mystere 20;;
- : int = 25
```

Application partielle d'une fonction

```
# let ajouteur x = function s -> s + x;;
val ajouteur : int -> int -> int = <fun>

# let ajouteur' x s = s + x;;
val ajouteur' : int -> int -> int = <fun>
```

Application partielle d'une fonction

```
# let ajouteur x = function s -> s + x;;
val ajouteur : int -> int -> int = <fun>

# let ajouteur' x s = s + x;;
val ajouteur' : int -> int -> int = <fun>

# ajouteur 3;;
- : int -> int = <fun>

# ajouteur' 3;;
- : int -> int = <fun>
```

Application partielle d'une fonction

```
# let ajouteur x = function s -> s + x;;
val ajouteur : int -> int -> int = <fun>

# let ajouteur' x s = s + x;;
val ajouteur' : int -> int -> int = <fun>

# ajouteur 3;;
- : int -> int = <fun>

# ajouteur' 3;;
- : int -> int = <fun>


# ajouteur 3 4;;
- : int = 7

# ajouteur' 3 4;;
- : int = 7
```

Sous-expression

```
# let a = "bonjour";;
val a : string = "bonjour"
```

Sous-expression

```
# let a = "bonjour";;
val a : string = "bonjour"

# let a = 10 in
    a + 20;;
- : int = 30
```

Sous-expression

```
# let a = "bonjour";;
val a : string = "bonjour"

# let a = 10 in
    a + 20;;
- : int = 30

# a;;
val a : string = "bonjour"
```

Sous-expression

```
# let a = "bonjour";;
val a : string = "bonjour"

# let a = 10 in
    a + 20;;
- : int = 30

# a;;
val a : string = "bonjour"
```

L'expression `let` α `in` **crée** un **sous-environnement** dans lequel :

- est enregistré la liaison α,
- est évaluée l'expression qui suit.

Sous-expression

```
# let a = "bonjour";;
val a : string = "bonjour"

# let a = 10 in
    a + 20;;
- : int = 30

# a;;
val a : string = "bonjour"
```

L'expression `let` α `in` **crée** un **sous-environnement** dans lequel :

- est enregistré la liaison α,
- est évaluée l'expression qui suit.

Syntaxe :

```
let nom_1 = expr_1 in
  let nom_2 = expr_2 in
    let nom_3 = expr_3 in
      expr_finale
```

Sous-expression

```
# let a = "bonjour";;
val a : string = "bonjour"

# let a = 10 in
    a + 20;;
- : int = 30

# a;;
val a : string = "bonjour"
```

L'expression `let` α `in` **crée** un **sous-environnement** dans lequel :

- est enregistré la liaison α,
- est évaluée l'expression qui suit.

Syntaxe :

```
let nom_1 = expr_1 in
  let nom_2 = expr_2 in
    let nom_3 = expr_3 in
      expr_finale
```

Avec des liaisons simultanées :

```
let nom_1 = expr_1
and nom_2 = expr_2
and nom_3 = expr_3 in
    expr_finale
```

Exemple courant d'utilisation d'une sous-expression

```
# let rec compte acc n = if n = 0 then acc
                         else compte (acc+1) (n-1);;
val compte : int -> int = <fun>

# compte 0 40_000_000;;
- : int = 40000000
```

Exemple courant d'utilisation d'une sous-expression

```
# let rec compte acc n = if n = 0 then acc
                         else compte (acc+1) (n-1);;
val compte : int -> int = <fun>

# compte 0 40_000_000;;
- : int = 40000000
```

il est dommage (et dangereux) de devoir ajouter un argument qui doit toujours valoir 0

Exemple courant d'utilisation d'une sous-expression

```
# let rec compte acc n = if n = 0 then acc
                         else compte (acc+1) (n-1);;
val compte : int -> int = <fun>

# compte 0 40_000_000;;
- : int = 40000000
```

il est dommage (et dangereux) de devoir ajouter un argument qui doit toujours valoir 0

Solution : créer une fonction auxiliaire **locale** :

```
# let compte n = let rec aux acc p =
                     if p = 0 then acc
                     else aux (acc+1) (p-1)
                 in aux 0 n;;
compte : int -> int = <fun>
```

La suite de Fibonacci

(si on a le temps)

La suite de Fibonacci est définie comme suit :

$F(0) = 0$

$F(1) = 1$

$F(n) = F(n-1) + F(n-2)$

L'algorithme de Luhn

(si on a le temps)

Pour vérifier qu'un numéro est « valide ».

un numéro (SIREN)	5	3	0	3	6	8	1	5	8
multiplier par	1	2	1	2	1	2	1	2	1
résultat	5	6	0	6	6	16	1	10	8
réduire	5	6	0	6	6	7	1	1	8
somme	S=40								

La somme `S` modulo 10 d'un numéro SIREN doit valoir 0.

Si `S mod 10 <> 0` alors il y a une erreur dans le numéro.

Inférence de type : la base

```
# 10+20;;
- : int = 30

# (+);;
- : int -> int -> int = <fun>

# 10.2 +. 20.3;;
- : float = 30.5

# 10 + 20.3;;
Error: This expression has type float but an expression was
expected of type int

# 10 +. 20;;
Error: This expression has type int but an expression was
expected of type float
```

Inférence de type : la base

Solution : conversion explicite :

```
# float 10;;
- : float = 10.

# int_of_float 23.4;;
- : int = 23

# string_of_int 45;;
- : string = "45"
```

`int_of_string`, `float_of_string`, `string_of_float`, etc.

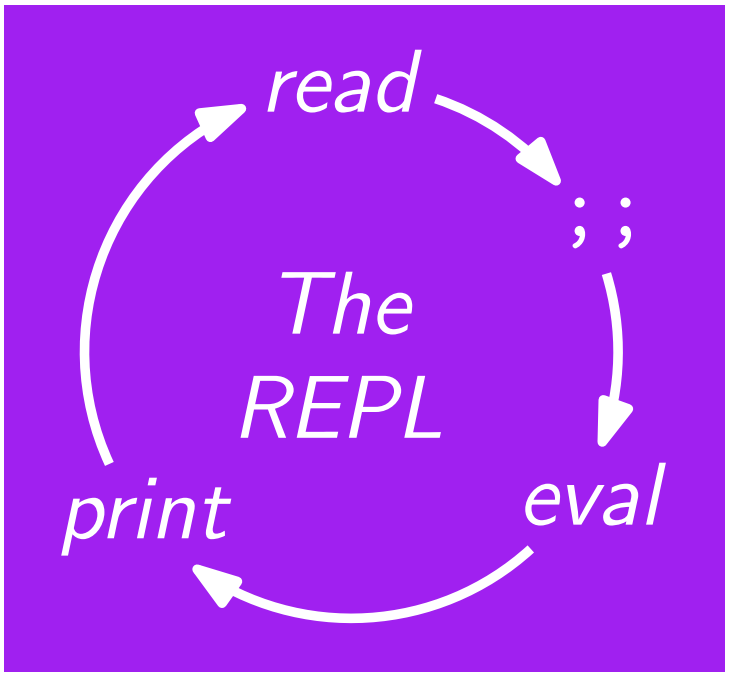
```
;;
```

et le `;;`, qu'est-ce que c'est ?

;;

et le `;;`, qu'est-ce que c'est ?

Il sert uniquement à la boucle interactive `ocaml` pour savoir quand commencer à interpréter ce que l'utilisateur écrit.

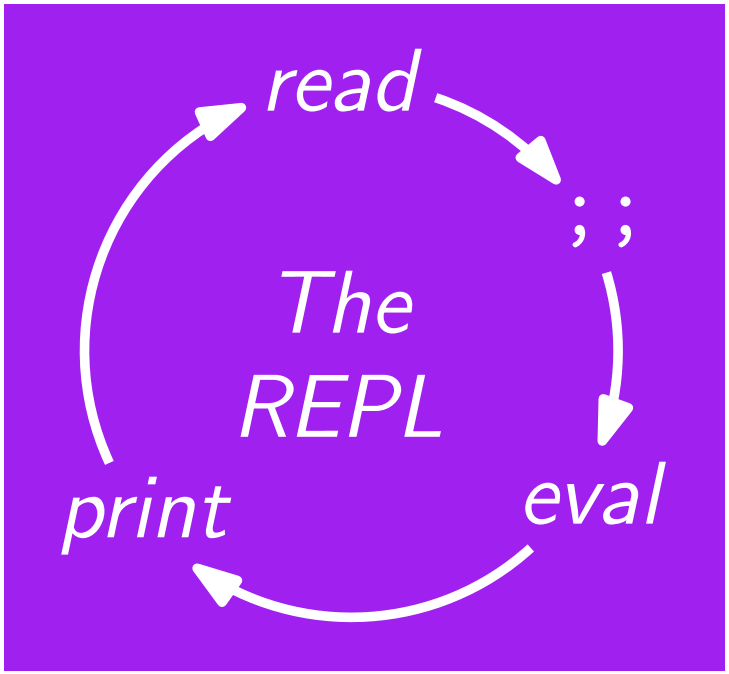


;;

et le `;;`, qu'est-ce que c'est ?

Il sert uniquement à la boucle interactive `ocaml` pour savoir quand commencer à interpréter ce que l'utilisateur écrit.

En interne, il est remplacé par `in` :
Quand on utilise la boucle interactive, on n'arrête pas d'imbriquer des expressions les unes dans les autres.



Récapitulatif

La boucle interactive d'Ocaml :

```
$ ocaml
print_endline string ;;
#quit ;;
^D
```

Types de base :

int	23 -139
float	4.2 -5.58e-2
bool	true false
char	'a' '4' '\n'
string	"bonjour le\tmonde" "retour chariot\n"
list	[45 ;26 ;-4 ;348]
tuple	(1,2) ('a', 18) ("Robert", 37, 1.82)
unit	()

Récapitulatif

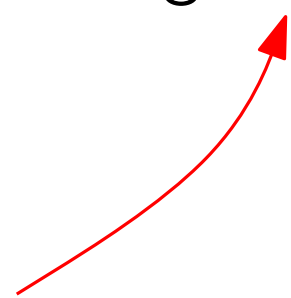
La boucle interactive d'Ocaml :

```
$ ocaml
print_endline string ;;
#quit ;;
^D
```

Types de base :

int	23 -139
float	4.2 -5.58e-2
bool	true false
char	'a' '4' '\n'
string	"bonjour le\tmonde" "retour chariot\n"
list	[45 ;26 ;-4 ;348]
tuple	(1,2) ('a', 18) ("Robert", 37, 1.82)
unit	()

Encodage source : ISO Latin 1



Récapitulatif

La boucle interactive d'Ocaml :

```
$ ocaml
print_endline string ;;
#quit ;;
^D
```

en français : n-uplet

Types de base :

int	23 -139
float	4.2 -5.58e-2
bool	true false
char	'a' '4' '\n'
string	"bonjour le\tmonde" "retour chariot\n"
list	[45 ;26 ;-4 ;348]
tuple	(1,2) ('a', 18) ("Robert", 37, 1.82)
unit	()

Encodage source : ISO Latin 1

Ocaml en bref

— typage statique **et** inférence automatique de type
— **programmation fonctionnelle, modulaire**, objet, impérative
— système d'exceptions
— gestion automatique de la mémoire (ramasse miettes)

Ocaml en bref

- typage statique **et** inférence automatique de type
- **programmation fonctionnelle, modulaire**, objet, impérative
- système d'exceptions
- gestion automatique de la mémoire (ramasse miettes)

Outils

interpréteur interactif : `ocaml`
en script : `#!/usr/bin/env ocaml`
compilateur vers machine virtuelle : `ocamlc`
machine virtuelle : `ocamlrun`
compilateur vers code machine : `ocamlopt`
`ocamldebug`, `ocamlprof`
etc.

Programmation fonctionnelle

La programmation fonctionnelle est

— un *style* de programmation

— une manière de programmer

mettant en avant la notion de *fonction* et d'*immutabilité*.

Programmation fonctionnelle

La programmation fonctionnelle est

— un *style* de programmation

— une manière de programmer

mettant en avant la notion de *fonction* et d'*immutabilité*.

1. Une fonction est vue comme une fonction mathématique :

même argument $\Rightarrow$ même résultat

Programmation fonctionnelle

La programmation fonctionnelle est

— un *style* de programmation

— une manière de programmer

mettant en avant la notion de *fonction* et d'*immutabilité*.

1. Une fonction est vue comme une fonction mathématique :

même argument $\Rightarrow$ même résultat

2. Les fonctions sont des valeurs que l'on peut manipuler tout comme d'autres type de base (entier,réel, caractère, chaîne de caractère, ...)

— on peut passer un fonction en paramètre d'une fonction

— une fonction peut retourner une fonction

OCaml en pratique

The Unison File Synchronizer (changement Java $\rightarrow$ OCaml au milieu du développement)

OCaml en pratique

The Unison File Synchronizer (changement Java $\rightarrow$ OCaml au milieu du développement)

The MLdonkey peer-to-peer client

OCaml en pratique

The Unison File Synchronizer (changement Java $\rightarrow$ OCaml au milieu du développement)

The MLdonkey peer-to-peer client

Industrie financière : *LexiFi*, *Jane Street* (trading)

OCaml en pratique

The Unison File Synchronizer (changement Java $\rightarrow$ OCaml au milieu du développement)

The MLdonkey peer-to-peer client

Industrie financière : *LexiFi*, *Jane Street* (trading)

The ASTRÉE Static Analyzer $\ll$ *So far, ASTRÉE has proved the absence of runtime errors in the primary control software of the Airbus A340 family* $\gg$

OCaml en pratique

The Unison File Synchronizer (changement Java $\rightarrow$ OCaml au milieu du développement)

The MLdonkey peer-to-peer client

Industrie financière : *LexiFi*, *Jane Street* (trading)

The ASTRÉE Static Analyzer « *So far, ASTRÉE has proved the absence of runtime errors in the primary control software of the Airbus A340 family* »

FFTW génère du code C optimisé pour les calculs de transformées de Fourier.

OCaml en pratique

The Unison File Synchronizer (changement Java $\rightarrow$ OCaml au milieu du développement)

The MLdonkey peer-to-peer client

Industrie financière : *LexiFi*, *Jane Street* (trading)

The ASTRÉE Static Analyzer « *So far, ASTRÉE has proved the absence of runtime errors in the primary control software of the Airbus A340 family* »

FFTW génère du code C optimisé pour les calculs de transformées de Fourier.

Vérification automatique de *driver* chez Microsoft

OCaml en pratique

The Unison File Synchronizer (changement Java $\rightarrow$ OCaml au milieu du développement)

The MLdonkey peer-to-peer client

Industrie financière : *LexiFi*, *Jane Street* (trading)

The ASTRÉE Static Analyzer « *So far, ASTRÉE has proved the absence of runtime errors in the primary control software of the Airbus A340 family* »

FFTW génère du code C optimisé pour les calculs de transformées de Fourier.

Vérification automatique de *driver* chez Microsoft

Facebook utilise OCaml (entre autres)

Programmation fonctionnelle en pratique

Erlang

- conçu par Ericsson pour système de télécommunication robuste
- fonction chat de *Facebook* (avec *ejabberd*)
- *Twitterfall*, analyse les tendances sur *Twitter*
- *Wings 3D*, un modeleur 3D

Programmation fonctionnelle en pratique

Erlang

- conçu par Ericsson pour système de télécommunication robuste
- fonction chat de *Facebook* (avec *ejabberd*)
- *Twitterfall*, analyse les tendances sur *Twitter*
- *Wings 3D*, un modeleur 3D

Haskell

- utilisé par plusieurs banques pour leurs logiciels internes
- *Facebook* : manipulation automatique de code PHP

Programmation fonctionnelle en pratique

Erlang

- conçu par Ericsson pour système de télécommunication robuste
- fonction chat de *Facebook* (avec *ejabberd*)
- *Twitterfall*, analyse les tendances sur *Twitter*
- *Wings 3D*, un modeleur 3D

Haskell

- utilisé par plusieurs banques pour leurs logiciels internes
- *Facebook* : manipulation automatique de code PHP

Autres langages fonctionnels : Scala, {Standard ML, F#}, {Lisp, Scheme, Clojure}

Fin

Fin du cours d'aujourd'hui.

Le prochain cours sera plus léger et portera sur les **types** et la création de nouveaux types.

http ://ocaml.org/

Garde dans un motif

On peut affiner un motif par l'ajout de conditions appelées **gardes** :

```
# let signe n = match n with
    | 0 -> "nul"
    | x when x < 0 -> "négatif"
    | _ -> "positif";;
val signe : int -> string = <fun>
```

Garde dans un motif

On peut affiner un motif par l'ajout de conditions appelées **gardes** :

```
# let signe n = match n with
    | 0 -> "nul"
    | x when x < 0 -> "négatif"
    | _ -> "positif";;
val signe : int -> string = <fun>
```

est identique à :

```
# let signe n = if n = 0 then "nul"
                else if n > 0 then "positif"
                     else "négatif";;
val signe : int -> string = <fun>
```

Garde dans un motif

On peut affiner un motif par l'ajout de conditions appelées **gardes** :

```
# let signe n = match n with
    | 0 -> "nul"
    | x when x < 0 -> "négatif"
    | _ -> "positif";;
val signe : int -> string = <fun>
```

ça s'appelle une `garde`

est identique à :

```
# let signe n = if n = 0 then "nul"
                else if n > 0 then "positif"
                     else "négatif";;
val signe : int -> string = <fun>
```

Liaisons simultanées

```
# let pair k = if k = 0 then true
                else impair (k-1);;
# let impair k = if k = 0 then false
                  else pair (k-1);;
Error: Unbound value impair
```

Liaisons simultanées

```
# let pair k = if k = 0 then true
               else impair (k-1);;
# let impair k = if k = 0 then false
                 else pair (k-1);;
Error: Unbound value impair
```

Solution : **liaison simultanée** avec le mot clé `and`

```
# let rec pair k = if k = 0 then true
                   else impair (k-1)
  and impair k = if k = 0 then false
                 else pair (k-1);;
val pair : int -> bool = <fun>
val impair : int -> bool = <fun>

# impair 22;;
- : bool = false
```

Récursivité terminale

```
# let rec compte n = if n <= 0 then 0
                     else 1 + compte (n-1);;
val compte : int -> int = <fun>

# compte 23;;
- : int = 23

# compte 1000;;
- : int = 1000

# compte 300_000;;
Stack overflow during evaluation (looping recursion?).
```

Récursivité terminale

```
# let rec compte_aux acc n = if n = 0 then acc
                             else compte_aux (acc+1) (n-1);;
val compte_aux : int -> int = <fun>

# let rec compte n = compte_aux 0 n;;
val compte : int -> int = <fun>

# compte 0 300_000;;
- : int = 300000

# compte 0 40_000_000;;
- : int = 40000000
```